# Understanding AOP through the Study of Interpreters

Robert E. Filman
Research Institute for Advanced Computer Science
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035
rfilman@mail.arc.nasa.gov

## ABSTRACT

I return to the question of what distinguishes AOP languages by considering how the interpreters of AOP languages differ from conventional interpreters. Key elements for static transformation are seen to be redefinition of the set and lookup operators in the interpretation of the language. This analysis also yields a definition of crosscutting in terms of interlacing of interpreter actions.

## 1. INTRODUCTION

I return to the question of what distinguishes AOP languages from others [12, 14].

A good way of understanding a programming language is by studying its interpreter [17]. This motif has been recently emphasized in recent work by Masuhara and Kiczales [21], and is a theme of the work on the Aspect Sandbox [22, 30]. This position paper suggests studying the foundations of Aspect-Oriented Languages by considering what changes have to be made to conventional language interpreters to get aspect behaviors.

Interpreters express meaning. Compilers can be understood as optimizations that enable more efficient renderings of the work of interpreters, without changing the underlying meaning of programs. A compiler that builds a "new program" from several fragments can be understood as a substitute for the dynamic, run-time building of that new program from fragments. Thus, while compilation techniques for AOP (e.g., partial evaluation as weaving [22]) are quite worthwhile activities, they do address the question of the nature of AOP.

## 2. A GENERIC INTERPRETER

Consider the pseudo-code for interpreter for A Generic Programming Language (AGPL) in Figure 1. This is a "meaning" function over an "expression" (an object in expression space), and an "environment," a structure that maps names to values, perhaps with a characterization of what kind of mapping is of interest (e.g., variables vs. functions). The pseudo-code includes only enough detail to convey the ideas I'm trying to express.

Of course, a real implementation would need implementations of the helper functions. In general, the helper functions on environments—lookup, set, and extend—can be manipulated to create a large variety of different language features. The most straightforward implementation makes an environment a set of symbol–value pairs (a map from symbols to values) joined to a pointer to a parent environment. Lookup finds the pair of the appropriate symbol, (perhaps chaining through the parent environments in its search), returning its value; set finds the pair and changes its value field; and extend builds a new map with some initial values whose parent is the environment being extended. In this model, lookup and set are "reference assignment" pairs: they act like elements setting and retrieving the value of a location. Programming languages vary by their use of chaining in environments. Most languages have some notions of global environment (a parent of or shared by all elements) and of constant elements (ones that don't change, such as a global function being assigned to a particular value.) Some languages may use the name being looked up or set as a structured object that guides the search in the environment space. More formal approaches would substitute a monad for the state expressed in the environment, but that level of formality would only obscure the discussion here.

I have provided set and lookup types (e.g., `VARIABLE` and `FUNCTION`) so that the implementations of set and lookup can be manipulated to separate things such as the function and variable space (as in Common Lisp [26]) or to conflate them (as in Scheme ([5]). By providing a richer notion of "set," one can create languages that export and restrict visibility; by providing a richer notion of "lookup" one can get inheritance. Most appropriate for doing independently created aspects (as opposed to aspects merely defined in the same "file") is the idea that certain varieties of environment.set change or extend some root (or at least non-leaf) environment.

Focusing on set and lookup corresponds to the importance of naming in practical programming languages. Much of the art of programming language design is the rules for associating names with meanings and groupings, and the visibility of these names; much of the act of programming is invoking named entities, dynamically associating names with values, and retrieving the values of names.

## 3. CROSSCUTTING AND BLAME

One can assign credit (or blame) to every external action (a primop) or manipulation in the interpreter. Each action in the interpreter is associated with a particular expression, the most immediate cause for that action. One can divide expressions into "modules." In general, actions follow the structure of expressions, and actions tend to proceed within a module. I call this overall notion of the corresponding continuity in the expression space and the action sequence *locality*.

We have *crosscutting* when sequences of actions intermix from different modules. In conventional languages crosscutting arises most often as explicit invocation: an expression in one module names an entry of another module, and the system transfers control to that other module. Some conventional languages allow other crosscutting mechanisms. For example, in languages with function pointers or dynamic binding, the value of a dynamic environmental element can be used as an expression for further evaluation. Inheritance mechanisms also combine the code of several modules (equivalent to modifying the lookup function to search parent environments). In some languages, type declarations can have the

```
/* Compute the meaning of an expression, exp, given a environment, env
```

<0> I cheat by using the stack of the machine interpreting meaning as the stack for meaning. A richer (and perhaps more appropriate) system can be build by maintaining our own stack, allowing searches within that stack for elements like catch/throw and dynamic calling scope.

<1> The meaning of a literal expression is the constant of the expression. Numerals, strings, and quoted expressions are literals.

<2> If exp is a variable, look up its meaning in the environment with respect to variable lookup.

<3> If exp is a primitive operator (one that executes on the underlying machine, like "plus" or "print,") evaluate the meanings of its arguments in the current environment, assemble them into a "value list," and invoke the primitive operator on that list.

<4> If exp is some form of language-explicit interpreter control (like an "if" or "switch" statement), compute the meaning of the condition of the expression, and then return the meaning of the appropriate choice element (like the "else part" or the "default case.")

<5> If exp is an assignment statement, change the environment appropriately. The assignmentType covers the varieties of assignments one might want to make—for example, assigning to a variable, defining a local function, defining the fields of a record, or defining a new global function.

<6> Call a function. Find the body associated with that function. Build a new environment, based on the original environment and perhaps some environmental information of the definition itself, which binds the formals of the called function to the values of the actual parameters, and compute the meaning of the body in this new environment. I could have generalized this a bit beyond call by value, but it's not worth the trouble for the ideas I'm trying to convey.

```
*/
    meaning (exp, env) =                                                                          /* <0> */
        typecase (exp) :
            literal     (exp) -> exp.literalValue                                                 /* <1> */
            variable    (exp) -> env.lookup (exp.variableName, 'VARIABLE)                         /* <2> */
            primop      (exp) -> apply (exp.primop, meaningList (exp.args, env))                  /* <3> */
            conditional (exp) -> meaning (exp.conditionChoice (meaning (exp.condition, env)), env) /* <4> */
            assignment  (exp) -> env.set (exp.variableName, meaning(exp.value, env), exp.assignmentType) /* <5> */
            funcall     (exp) -> let definition = env.lookup (exp.functor, 'FUNCTION)             /* <6> */
                                 in meaning (definition.body,
                                             env.extend (definition.formals,
                                                         definition.environment,
                                                         meaningList (exp.args, env)))
```

**Figure 1: AGPL interpreter**

effect of remotely modifying behavior. (Such mechanisms lie between the explicit invocation of a Fortran subroutine call and AOP.) Exception generation and handling can cause jumps in the execution sequence. One can also define the system in a "feature specific" manner, so that user-supplied code always runs in some specific circumstance. These latter mechanisms cause crosscutting.

The novelty of AOP is that the crosscutting mechanisms are implicit (oblivious) and general-purpose. That is, examination of the source code doesn't indicate that the crosscutting takes place. Instead, some external mechanism performs the surgery on the execution process. Modern AOP demands that the crosscutting mechanism be "general purpose," allowing modifying any code with respect to the structure of that code, not just a particular semantics. (This contrasts with some of the earlier special-purpose "aspect" languages [20].) Thus, a system that allows the user to define, say, "security code" to be invoked in particular contexts is a framework, not an AOP language.

# 4. MODIFYING THE INTERPRETER

The purpose of this exercise is to ask what does one have to do to make AGPL aspect-oriented? Here we are concerned with general aspect behavior, not a hook for solving a particular problem. That is, we want to be able to invoke arbitrary user code at joint points, not merely a selection from some predefined or parameterized behaviors.

We first note that modifying the interpreter for the specific requirements of a particular aspect language can always yield any (implementable) aspect language. Most generally this is true because any implemented aspect language has an interpreter. More specifically, every aspect language defines certain elements or events as joint points, places where it is possible to associate aspect behavior with the underlying code. We can change the interpreter to pause at every such join point and consult the (perhaps dynamic) dictionary of current aspects to see which apply. (And, as many have observed, "Anything you can do I can do meta"—in a meta-interpreter architecture, we can delay to the meta level the decision about whether each execution point is a join point [4, 27].) Given a rich enough language for describing the desired aspect conditions, determining the places that need modification (effectively, the shadow points in the program or the execution points of such shadows in the interpreter) may be an interesting problem [15, 22].

The problem with such an analysis is that changing the body of the interpreter is the way to implement any conceivable language. We'd prefer to restrict the changes to more neatly describe the aspect space. More specifically, the problem is not so much describing mechanisms to implement aspect languages but, ideally, mechanisms that implement only aspect languages, or, more realistically, mechanisms whose parameterization approximates the space of aspect languages.

## 4.1 Advising a function

More than one research group has provided its interpretation of how best to implement AOP. Perhaps the most primitive mechanism, common to most approaches is "advice" (wrapping) [28]. With advice, the definition of a function is embedded inside other behavior, which can execute before, after, or around the original function. Systems that allow wrapping include Composition Filters [3], OIF [13], AspectJ [18], and JAC [25]. A structurally consistent way to get advice is to change the definition of functions to include advice. To advise a single function $F$ with advice $A$, creating $A(F)$, we could find the pair that joins $F$ to its definition, and replace its value by $A(F)$.

More commonly, we want to advise not one function, but an entire set of them, particularly the ones that pass some predicate test. That is, we want to quantify over the function

space. An AOP system can be built with either an *open-world* or *closed-world* assumption. Closed world systems know at the start of execution all the code that might run in the system. Thus, a closed-world system could implement quantified advice by finding all the function definitions and redefining the ones that need the advice. An open-world system can dynamically acquire new code. In an open-world system, we also need to modify environment.set so that function definition and redefinition work with the advice mechanism—defining or redefining an advice-worthy function, must make the setting include the advice.

Note that there is also a natural symmetry between set and lookup. Anything one imagines doing at "set" time can be done at "lookup" time, so long as sufficient information is retained to perform the action.

### 4.2 Advising a field

Some AOP approaches (e.g., Hyper/J [24]) treat object fields as combinations of other elements. For example, one has the ability to externally state that field $f$ in object $r$ is to be the same as field $f'$ in object $r'$ when $r$ and $r'$ are regarded as parts of the definition of the same object, or that $f$ in $r$ and $f$ in $r'$ are not the same, even when $r$ is merged with $r'$. Treating a variable as a combination of other elements in some sense, is symmetric to the functional advice problem. With functional advice, we are working in function space and know only a few combinators (e.g., before, after, and around), though others are easy to imagine (for example, consider mixins in Flavors [23]). With variables, we're working in variable space, and can think of a variety of combinators—for example, the "same as" and "different" examples, above, "union" for set-valued fields, "append" for sequence valued ones, and so forth.

### 4.3 Program transformation

Several authors have argued for doing AOP by program transformation [6, 11, 15, 16, 19]. From the point of view of an interpreter, program transformation can be realized by performing the transformation steps as part of the function definition process. (This is, of course, a somewhat heavy-handed interpretation of transformation.)

### 4.4 Frameworks

Frameworks (e.g., [8]) combine functional wrapping with wrappers specific to framework decision points. This can be seen as a structured step in function assignment. However, frameworks more naturally resemble modifying the interpreter to the special case doing additional behavior on function calling.

### 4.5 Field and method insertion

Some AOP approaches (e.g., [18]) allow the introduction of additional fields and methods. Once again, these are examples of changing the semantics of environment setting.

### 4.6 Dynamic flow

There have been several proposals for aspects that pay attention to the dynamic behavior of program execution. For example, aspect invocation in AspectJ can be predicated on what's in the calling history (cflow) [18]. At the first FOAL workshop, we argued for generally treating AOP as generically reacting to execution events [15], a theme also expressed by others [7, 10, 9, 16, 29]. The effects of such proposals are more problematic for interpreter transformation. Cflow can be accommodated if we create our own stack for the interpreter, rather than using the implicit stack of the system executing the interpreter and search that stack at appropriate join points. Alternatively we could change the definitions of functions to leave appropriate markers lying around to be recognized at the right instants. These require some structural changes to the interpreter. Similarly, event reaction can be seen to be requiring pervasive interpreter change.

## 5. CLOSING REMARKS

In this position paper, I've explored the idea that the changes required in "ordinary" interpreters to realize AOP languages reveals elements about the essence of AOP languages. Many (particularly the static varieties) of AOP mechanisms can be seen as redefinition of the storage or retrieval actions in the interpreter, often at record and method definition time. Join point definitions that span multiple locations require the definition, storage or retrieval mechanisms to "quantify" over the space of candidate points. I've also defined crosscutting in terms of the mixture of modules causing actions to execute, and identified AOP with that crosscutting that lacks explicit or implicit mention in the module code.

## 6. REFERENCES


[1] *Workshop on Advanced Separation of Concerns (ECOOP 2001)*, June 2001.

[2] *FOAL 2002: Foundations of Aspect-Oriented Langauges (AOSD-2002)*, Mar. 2002.

[3] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, Oct. 2001.

[4] N. M. N. Bouraqadi-Saâdanii and T. Ledoux. How to weave? In *Workshop on Advanced Separation of Concerns (ECOOP 2001)* [1].

[5] W. Clinger and J. Rees. Revised4 report on the algorithmic programming language scheme. *LiSP Pointers*, 4(3), 1991.

[6] G. A. Cohen. Recombing concerns: Experience with transformation. In *Workshop on Multi-Dimensional Separation of Concerns (OOPSLA 1999)*, Nov. 1999.

[7] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. 27th ACM Symp. on Principles of Programming Languages*, pages 54–66, Jan. 2000.

[8] C. A. Constantinides, T. Elrad, and M. Fayad. Extending the object model to provide explicit support for crosscutting concerns. *Software Practice and Experience*, 32(7):703–734, May 2002.

[9] K. De Volder, J. Brichau, K. Mens, and T. D'Hondt. Logic meta-programming, a framework for domain-specific aspect programming languages. http://www.cs.ubc.ca/kdvolder/binaries/cacm-aop-paper.pdf.

[10] K. De Volder and T. D'Hondt. Aspect-oriented logic meta programming. In P. Cointe, editor, *Meta-Level Architectures and Reflection, 2nd International Conference on Reflection*, volume 1616 of *LNCS*, pages 250–272. Springer Verlag, 1999.

[11] K. De Volder, T. Tourwé, and J. Brichau. Logic meta programming as a tool for separation of concerns. In *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.

[12] R. E. Filman. What is aspect-oriented programming, revisited. In *Workshop on Advanced Separation of Concerns (ECOOP 2001)* [1].

[13] R. E. Filman, S. Barrett, D. D. Lee, and T. Linden. Inserting ilities by controlling communications. *Comm. ACM*, 45(1):116–122, Jan. 2002.

[14] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

[15] R. E. Filman and K. Havelund. Source-code instrumentation and quantification of events. In AOSD-FOAL02 [2], pages 45–49.

[16] P. Fradet and M. Südholt. AOP: Towards a generic framework using program transformation and analysis. In *Workshop on Aspect Oriented Programming (ECOOP 1998)*, June 1998.

[17] D. P. Friedman, C. T. Haynes, and M. Wand. *Essentials of programming languages (2nd ed.)*. Massachusetts Institute of Technology, 2001.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Comm. ACM*, 44(10):59–65, Oct. 2001.

[19] G. Kniesel, P. Costanza, and M. Austermann. JMangler—a framework for load-time transformation of Java class files. In *First IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, Nov. 2001.

[20] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[21] H. Masuhara and G. Kiczales. A modeling framework for aspect-oriented mechanisms; draft. http://www.cs.ubc.ca/

[22] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In AOSD-FOAL02 [2], pages 17–26.

[23] D. A. Moon. Object-oriented programming with flavors. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–8. ACM Press, Nov. 1986.

[24] H. Ossher and P. Tarr. The shape of things to come: Using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. *Comm. ACM*, 44(10):43–50, Oct. 2001.

[25] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192*, pages 1–24. Springer-Verlag, Sept. 2001.

[26] G. Steele Jr. *Common Lisp: The Language, 2nd Edition*. Digital Press, Bedford, Massachusetts, 1990.

[27] G. T. Sullivan. Aspect-oriented programming using reflection and meta-object protocols. *Comm. ACM*, 44(10):95–97, Oct. 2001.

[28] W. Teitelman and L. Masinter. The Interlisp programming environment. *Computer*, 14(4):25–34, Apr. 1981.

[29] R. J. Walker and G. C. Murphy. Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. In *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*, May 2001.

[30] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In AOSD-FOAL02 [2], pages 1–8.